Scientia Iranica D (2012) 19 (3), 627–634

Sharif University of Technology

Scientia Iranica

Transactions D: Computer Science & Engineering and Electrical Engineering

www.sciencedirect.com

A software pipelining algorithm of streaming applications with low buffer requirements

A. Hatanaka *, **N. Bagherzadeh**

Department of Electrical Engineering and Computer Science, University of California, Irvine, 2200 Engineering Hall, Irvine, CA 92697-2625, USA

Received 26 August 2010; revised 1 June 2011; accepted 20 August 2011

KEYWORDS

Software pipelining;
Streaming;
Scheduling;
DMA;
Mixed integer linear.

Abstract Stream programming languages have become popular owing to their representations that enable parallelization of applications via static analysis. Several research groups have proposed approaches to software pipeline streaming applications onto multi/many-core architectures, such as CELL BE processors and NVIDIA GPUs. In this paper, we present a novel scheduling algorithm that software-pipelines streaming applications onto multi/many core architectures. The algorithm generates software pipeline schedules by formulating and solving MILP (Mixed Integer Linear Programming) problems. Experimental results show that compared to previous works, our approach generates schedules that use up to a 71% smaller amount of buffers needed for communication between kernels.

1. Introduction

The advent of multicore architectures has forced the software industry to rethink how software should be written. The programming styles adopted in the sequential computing era are no longer suitable for the new multicore architectures. The industry is in search of programming models and runtime systems that effectively exploit the computational power of multicore processors.

One programming model that is suitable for multicore platforms is the streaming programming model. Although there is a variety of models of computations [1,2] for streaming applications, as well as programming languages and frameworks built on top of them [3–5], many of them can be represented as computational kernels or processes sending or receiving data to or from each other, through communication channels. The advantage of this representation is that it enables compilers or run time systems to devise efficient schedules using the information about communications and dependencies between kernels.

In this paper, we present a software pipelining algorithm that schedules streaming applications onto multicore architectures with the goal of suppressing the amount of buffers used. Reducing the amount of buffers enables running an application on a lower-cost hardware.

The algorithm consists of two steps, namely, partitioning and scheduling, both of which are formulated and solved as MILP problems. By synchronizing producer and consumer kernel pairs more frequently, the algorithm generates schedules that have substantially lower buffer requirements and shorter makespans compared to previous works.

2. Streaming applications and target architecture of this work

Streaming applications in this paper are modeled as a variant of Synchronous Dataflow [2]. The graph representation of sample stream application is shown in Figure 1. Communication channels are connected to kernels via ports. Multicasting can be expressed as multiple channels originating from the same port. The amount of data transferred through a port per firing of a kernel is constant throughout the execution of the application. The software pipelining algorithm described later accepts only the applications which have kernels of uniform firing rates. Therefore, some of the kernels in a Synchronous Dataflow application may have to be replicated to resolve mismatches in data rates of input and output ports before the application is fed to the algorithm.

* Corresponding author.
E-mail addresses: ahatanak@uci.edu (A. Hatanaka), nader@uci.edu (N. Bagherzadeh).

Peer review under responsibility of Sharif University of Technology.

Production and hosting by Elsevier

1026-3098 © 2012 Sharif University of Technology. Production and hosting by Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.scient.2011.08.034

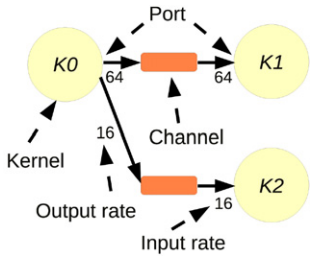


Figure 1: Stream graph example.

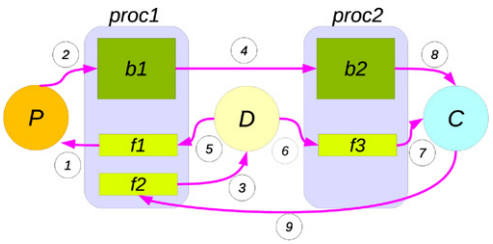


Figure 2: Interprocessor communication.

The target architecture consists of multiple processors that communicate with each other through an interconnection network. It is assumed that *DMA* hardware is attached to each processor in the architecture, which transfers data to or from other processors without intervention of the processor. There are two types of *DMA* request: *DMA* 'send' and 'receive'. *DMA* send requests are handled in a non-blocking manner. The processor continues with its computation after notifying the send request to the *DMA* hardware. *DMA* read requests are handled in a blocking manner.

The partitioning and scheduling algorithms presented later do not assume any specific interconnection networks between the processors in the architecture. However, it is assumed that the *DMA* transfer time is uniform. That is, regardless of which processor sends or receives data, the time it takes to set up the *DMA* transfer, send data over the network and receive it is the same.

Synchronization mechanisms resembling signals used in CELL processors [6] are used for communication between kernels mapped to different processors. Figure 2 is an example of how an interprocessor communication between a pair of producer and consumer kernels works.

P on processor $proc1$ and C on processor $proc2$ are the producer and consumer kernels, respectively. P writes the data it has produced into buffer $b1$ on $proc1$. Then, the *DMA* transfer, D, sends data in buffer $b1$ to buffer $b2$ on $proc2$, from which C reads. $f1$, $f2$ and $f3$ are the flags bits that indicate the presence or absence of valid data in the corresponding buffers. A *DMA* transfer is always initiated right after the completion of a producer kernel.

The following is the protocol of interprocessor communication via *DMA*.

1. Producer P first checks flag $f1$ to see if data in buffer $b1$ has already been consumed by *DMA* transfer, D. If it is not, P needs to wait till $f1$ is cleared.
2. Once $f1$ is cleared and buffer $b1$ is free, P starts and writes data into $b1$.
3. *DMA* transfer, D, follows the completion of P. D checks flag $f2$ to see if data in buffer $b2$ has been consumed by consumer C.
4. Once $f2$ is cleared and buffer $b2$ is free, D starts sending data in buffer $b1$ over the communication network to write it into buffer $b2$.
5. When all data in buffer $b1$ is sent out, D sets flag $f1$ to notify P of the completion of the *DMA* transfer.
6. Flag $f3$ is also set, so that C knows data in $b2$ is ready to be consumed.
7. Consumer C keeps waiting for flag $f3$ to be set by D.
8. When $f3$ is set, C starts reading and processing data in buffer $b2$.
9. Once C completes and data in $b2$ is no longer needed, it sets flag $f2$ to inform D in the next iteration that $b2$ is free.

3. Software pipelining algorithm

3.1. Background and terminologies

The basic idea behind software pipelining is the same, regardless of whether a streaming application is mapped to a multicore architecture, or a general purpose sequential program is compiled onto an instruction level parallel architecture. As such, software pipelining of streaming applications shares common terminologies and concepts with that of traditional software pipelining [7,8].

Figure 3 shows an example of a streaming application consisting of four kernels software pipelined onto an architecture consisting of two processors, $p0$ and $p1$. The nodes, $d0$ and $d2$, represent *DMA* transfers. The numbers below the nodes of the stream graph represent the execution time of the kernels. T, which is 5 in the example, is the initiation interval. The schedule is divided into stages, each of which has the length equal to T. Makespan refers to the length of a single iteration of a schedule: it is 25 in the example.

3.2. Buffer usage computation

The goal of the software pipelining algorithm in this paper is to minimize the usage of local memory buffers while meeting the provided throughput constraint. Therefore, it is important to analyze how buffers are used by different types of communication and determine the number of buffers needed for each of them. The buffers used for communication between kernels are classified into three categories, namely, intraprocessor buffers, producer-*DMA* buffers and *DMA*-consumer buffers.

Intraprocessor buffers are used for communication between kernels mapped to the same processor. The number of buffers needed can be determined simply by computing the interval between the start time of the producer and the end time of the consumer. Figure 4 is an example of a pair of producer-consumer kernels, *Prod* and *Cons*, mapped to the same processor, *Proc*0. In Figure 4(a), *Cons*(0), the first iteration of *Cons*, completes execution before *Prod*(1), the second iteration of *prod*, starts. Therefore, only one buffer is needed, since the buffer used for communication between the first iteration of *Prod* and *Cons* can be used in the next iteration and all the iterations after that. On the other hand, in Figure 4(b), *Prod*(2), the third iteration of *Prod*, starts executing before the data written by *Prod*(1) is consumed by *Cons*(1). In order to prevent the data written by *Prod*(1) from being overwritten by *Prod*(2), another buffer needs to be allocated. In general, the number of buffers needed, N, is determined by the following inequalities:

$$(N-1) \times T \leq e(C_i) - s(P_i) < N \times T \quad (1)$$

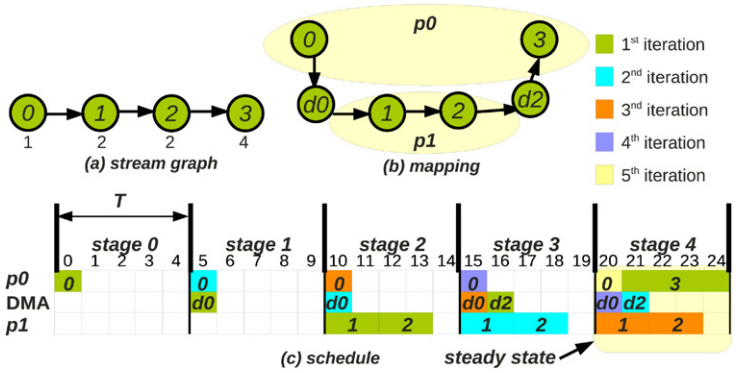
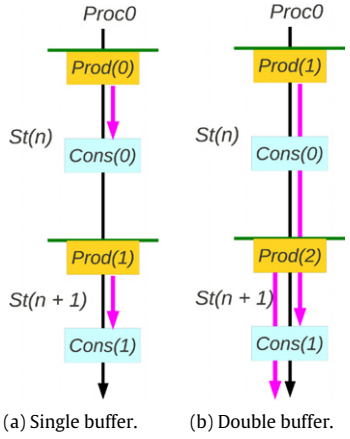


Figure 3: Example of software-pipelined streaming application.



(a) Single buffer. (b) Double buffer.

Figure 4: Intraprocessor communication buffer usage.

$e(K)$ and $s(k)$ are the end time and start time of kernel K, and C_i and P_i are the consumer and producer kernels of the ith iteration.

Producer-*DMA* buffers are used by a producer kernel and the *DMA* transfer process to send the data written by the producer kernel to consumer kernels mapped to other processors. The number of buffers needed for producer-*DMA* buffers can be calculated similarly to Eq. (1).

$$(N-1)\times T \le e(DMA_i) - s(P_i) < N \times T. \quad (2)$$

Since the *DMA* process initiates right after completion of the producer process, the inequality above can be rewritten as follows:

$$(N-1)\times T \le et(P) + t_{DMA} < N \times T \quad (3)$$

$et(K)$ is the execution time of kernel K and t_{DMA} is the interval between the time a *DMA* transfer starts and completes writing to the buffer on the consumer kernel's processor.

Buffers used for interprocesssor communications and located on the consumer processor are called *DMA*-consumer buffers. A *DMA* transfer process must make sure the consumer has completed reading the content of a buffer before it starts sending out data and overwrites it. As explained in Figure 2, this requires *DMA* transfer processes and consumer kernels to send synchronization signals to each other in addition to the payload data. A *DMA* transfer process sends a signal to inform the consumer that it has completed writing the data the consumer needs to a buffer, and the consumer sends back an acknowledgment signal to inform the *DMA* process that data has been read and the buffer is empty. Figure 5 is a timing diagram of a producer kernel sending data to a consumer kernel via *DMA*. In the diagram, t_{Ack} is the time it takes for the acknowledgment signal to reach the processor that initiated the *DMA* transfer. In Figure 5(a), only one buffer is needed on the consumer processor, since the acknowledgment signal sent from *Cons*(0) reaches processor *Proc*0 before *DMA*(1) starts sending out data for the next iteration. In Figure 5(b), the acknowledgment signal does not arrive in time and therefore another buffer is needed.

The number of buffers needed for *DMA*-consumer communication is determined by the following inequality:

$$(N-1)\times T \le (e(C_i) + t_{Ack}) - s(DMA_i) < N \times T. \quad (4)$$

The start time of a *DMA* transfer process and the end time of its producer kernel are equal. Therefore:

$$(N-1)\times T \le (e(C_i) + t_{Ack}) - e(P_i) < N \times T. \quad (5)$$

3.3. Algorithm

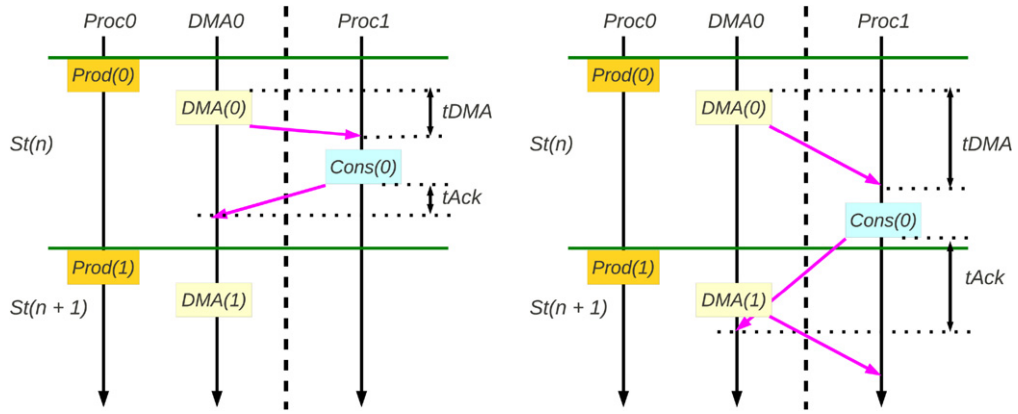
3.3.1. Overview

The software pipelining algorithm of this paper takes, as input, the stream graph of the application, architecture parameters, such as the number of processors in the architecture, *DMA* transfer time and the initiation interval. The output is the kernel to processor mapping information and the start time and end time of the kernels. The algorithm consists of two steps: partitioning and scheduling. Both steps are formulated and solved as MILP problems.

The partitioning step decides the mapping from kernels to processors. The goal of this step is to find a partitioning of kernels that minimizes the largest amount of buffer space processors need. The precise amount of buffers needed cannot be computed at the partitioning step, since the start and end times of kernels are not fixed yet. The partitioning step focuses on reducing the amount of buffers needed by interprocessor communication, expecting that this will lead to a reduction in the overall amount of buffers.

The scheduling step takes, as input, the solution obtained in the partitioning step and assigns start and end times to the kernels. Since it is known whether a communication between two kernels is interprocessor or intraprocessor, the number of buffers needed can be computed using the formula presented in Section 3.2. Using the information available, the scheduling step produces a software pipelined schedule that minimizes the largest buffer usage, while achieving the throughput determined by the initiation interval.

The remainder of this section presents the parameters, variables, constraints and objective functions of the MILP problems for the partitioning and scheduling steps.



(a) Single buffer. (b) Double buffer.

Figure 5: Interprocessor communication buffer usage.

3.3.2. Partitioning step

- **Parameters.**
  - K. Set of kernel instances.
  - KT. Set of kernel types. Each kernel instance has its own kernel type. Two different kernel instances of the same kernel type have the same program code.
  - C. Set of channels. A channel can be expressed as two pairs of a kernel and a port number. For example, $((K0, 1), (K1, 2))$ is a channel that originates from output port 1 of kernel $K0$ and terminates at input port 2 of kernel $K1$.
  - CS. Set of source ports. For example, the source port of channel $((K0, 1), (K1, 2))$ is $(K0, 1)$.
  - P. Set of processors of the target architecture.
  - T. Initiation interval of the schedule.
  - $et(i \in KT)$. The average execution time of kernel type i. This can be obtained by profiling the kernel running on a processor of the target architecture or by static program analysis.
  - $kt(i \in K)$. The kernel type of kernel i.
  - $ktsize(i \in KT)$. The program code size of kernel type i.
  - $cssize(i \in CS)$. The amount of data sent through source port i during a single iteration.
  - $cs2dstk(i \in CS)$. The set of kernels which read data originating from source port i. For example, if there are two channels, $((K0, 0), (K1, 2))$ and $((K0, 0), (K2, 1))$, that originate from $(K0, 0)$, then $cs2dstk((K0, 0)) = \{K1, K2\}$.
- **Variables.**
  - $kp_{i,j}, i \in P, j \in K$. A binary variable set to 1 if kernel j is placed on processor i.
  - $ktp_{i,j}, i \in P, j \in KT$. A binary variable set to 1 if at least one instance of kernel type j is placed on processor i.
  - $dma_{i,j}, i \in P, j \in CS$. A binary variable. It is set to 1 if there is an interprocessor communication that terminates at processor i and originates from source port j. For example, if $K0$ and $K1$ are mapped to processor $proc0$ and $proc1$, respectively, and there is a channel $((K0, 0), (K1, 1))$, then $dma_{proc1,(K0,0)}$ is 1.
  - $lmd_i, i \in P$. The sum of the amount of data sent to processor i via DMA during a single iteration. For example, suppose there are just two channels, $((K0, 0), (K1, 1))$ and $((K2, 0), (K3, 1))$, and $cssize((K0, 0)) = 20$ and $cssize((K2, 0)) = 10$. If both $K1$ and $K3$ are mapped to processor $proc1$, then, $lmd_{proc1} = cssize((K0, 0)) + cssize((K2, 0)) = 30$.
  - $lmc_i, i \in P$. The sum of the program code size of the kernels mapped to processor i.
  - $lm_i, i \in P$. The total amount of buffer space on processor i that is used for data and program code.
  - max lm. The largest among lm_i.
  - $totallmd$. The total amount of data sent via interprocessor communication.
- **Constraints.**
  - A kernel must be mapped to exactly one processor. Therefore:

$$\sum_{i \in P} kp_{i,j} = 1, \quad \forall j \in K. \tag{6}$$

  - The sum of the execution time of kernels mapped to a processor must not exceed the provided initiation interval.

$$\sum_{k \in K} et(kt(k)) * kp_{i,k} <= T, \quad \forall i \in P. \tag{7}$$

  - The sum of data sent via DMA transfers is expressed as follows:

$$totallmd = \sum_{i \in P} lmd_i. \tag{8}$$

  - The amount of buffer needed for storing data cannot be computed precisely until the scheduling step is completed after the partitioning step. In order to deal with this phase ordering issue, the sum of the size of buffers used for data on processor i is approximated as lmd_i. Although the actual sum that is computed after the completion of the scheduling step will most likely be larger, using lmd_i is good enough as an approximate. Reducing the amount of buffers used for interprocessor communications will likely accomplish the purpose of steering the MILP solver to generate a solution with a small maximum buffer size. The constraints related to lm_i and max lm are as follows:

$$\max lm \geq lm_i, \quad \forall i \in P, \tag{9}$$

$$lm_i = lmd_i + lmc_i, \quad \forall i \in P, \tag{10}$$

$$lmd_i = \sum_{j \in CS} dma_{i,j} * cssize(j), \quad \forall i \in P, \tag{11}$$

$$lmc_i = \sum_{j \in KT} ktp_{i,j} * ktsize(j), \quad \forall i \in P. \tag{12}$$

It is worth noting that Constraint (11) takes into account the fact that *DMA*-consumer buffers can be shared among channels originating from the same source port. It guides the solver to map kernels that consume data from the same source port to the same processor.

– A source port j needs buffer space for interprocessor communication on processor i, if the producer kernel is not mapped to i and at least one of the consumer kernels of j is mapped to i. Therefore, the following constraint is added:

$$dma_{i,j} = \neg kp_{i,cssrc(j)} \wedge (\bigvee_{k \in cs2dstk(j)} kp_{i,k}), \quad \forall i \in P,\ \forall j \in CS. \tag{13}$$

Standard techniques are used to convert logical operators, $\wedge$ and $\vee$, to linear programming formats.

– At least one kernel of type j must be mapped to processor i in order for $ktp_{i,j}$ to be 1.

$$ktp_{i,j} = \bigvee_{\{k \in K : kt(k)=j\}} kp_{i,k}, \quad \forall i \in P,\ \forall j \in KT. \tag{14}$$

- **Objective function**. The objective is to minimize the sum of the total amount of *DMA* communication and the maximum size of buffer space.

$$\text{minimize}: \ totallmd + \max lm * |P|. \tag{15}$$

3.3.3. Scheduling step

- **Parameters.** The following parameters are used in the scheduling step, in addition to those used in the partitioning step.
  - $proc(i \in K)$. Mapping from a kernel, K, to a processor of the target architecture. This is generated by the partitioning step.
  - $cssrc(i \in CS)$. Source kernel of source port i.
  - $c2cs(i \in C), c2src(i \in C), c2dst(i \in C)$. Source port, source kernel and destination kernel of channel i, respectively.
  - $t_{DMA}(i \in CS)$. The *DMA* transfer time. This is the interval between the start time of a *DMA* transfer and the time it completes writing to the destination processor's buffer.
  - $t_{Ack}(i \in C)$. The time it takes for an acknowledgment signal to reach the destination processor.
- **Variables.**
  - $s_i, e_i, i \in K$. Start and end times of kernel i, respectively.
  - $N_i, i \in K$. The number of initiation intervals elapsed before the first execution of kernel i.
  - $off_i, i \in K$. Offset of kernel i.
  - $off\,0_i, i \in P$. External offset of kernels mapped to processor $i, \geq 0$.
  - $off\,1_i, i \in K$. Internal offset of kernel $i, \geq 0$.
  - $bC_{i,j}, i \in P, j \in C$. Number of buffers processor i needs for channel j.
  - $bCS_{i,j}, i \in P, j \in CS$. Number of buffers processor i needs for source port j.
  - $bDMA_{i,j}, i \in P, j \in CS$. Number of buffers processor i needs for *DMA* transfer to other processors originating from source port j.
  - $\min b_{i,j}$. Minimum number of buffers processor i needs for source port j.
  - lm_i. Total amount of local memory needed on processor i.
  - $\max lm$. The largest among lm_i.
- **Constraints.**
  - The constraint for start and end times of a kernel.

$$e_i = s_i + et(kt(i)), \quad \forall i \in K. \tag{16}$$

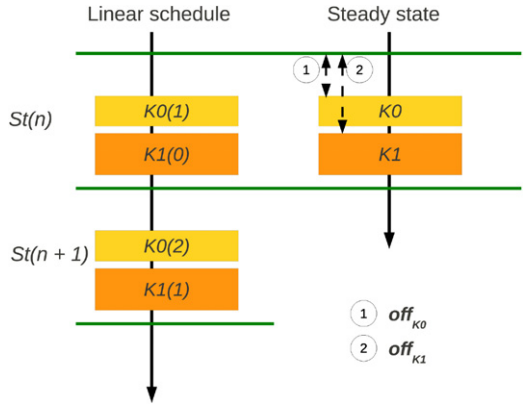


Figure 6: Schedule without kernels wrapping around.

– Dependency constraints are expressed as follows. $S4$ is a set of channels whose source and destination kernels are mapped on the same processor.

$$e(c2src(c)) <= s(c2dst(c)), \quad c \in S4, \tag{17}$$

$$e(c2src(c)) + t_{DMA}(c2cs(c)) <= s(c2dst(c)), \quad c \in \{C - S4\}. \tag{18}$$

– The kernels in a software pipelined schedule form a steady state which is repeatedly executed at a rate determined by the initiation interval. Therefore, constraints must be added that ensure kernels in the steady state mapped to the same processor do not overlap with each other. The constraints are easy to formulate if there are no kernels that wrap around stage boundaries, or in other words, have start and end times that belong to different stages. An example of such a schedule of kernels is shown in Figure 6. In this case, the constraint to prevent conflict between two kernels, i and j, mapped to the same processor is given as follows.

$$\begin{aligned} off_i &>= off_j + et(j) \vee, \\ off_j &>= off_i + et(i) \end{aligned} \tag{19}$$

off_i and off_j are the offsets of kernel i and j and can be computed as $off_i = s_i \% T$ and $off_j = s_j \% T$.

Formulating the constraints becomes more complicated if a case in which at least one of the kernels wraps around a stage boundary is considered. In such a case, inequalities in Constraint (19) are no longer sufficient to guarantee that two kernels do not conflict with each other in the steady state. Using the example in Figure 7, suppose kernel $K1$'s execution time was extended until the end time of $K1(0)$, which is larger than the start time of $K0(2)$. In that case, although the two kernels overlap in the steady state, the two kernels are considered conflict-free, according to Constraint (19), since $off_{K1} >= off_{K0} + et(K0)$ is satisfied.

In order to simplify the MILP formulation, a conceptual boundary box along with two new offset variables are introduced, as shown in the right side of Figure 7. A boundary box has a height equal to T, and encloses all the kernels in the steady state: There are no kernels that wrap around the boundary created by the bounding box. Each processors has its own boundary box $off\,0_{proc1}$ is the external offset of the boundary box of processor $proc1$, and is equal to the interval between the stage boundary and the top of the boundary box. $off\,1_K$ is the internal offset

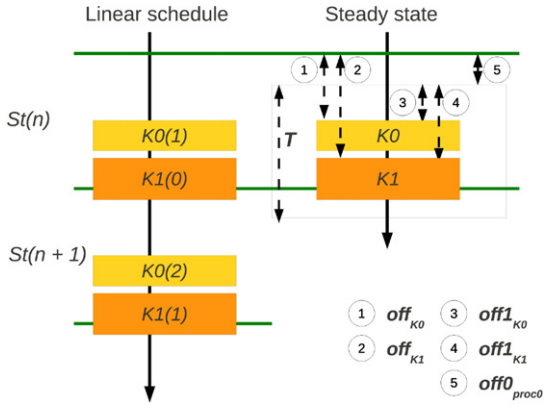


Figure 7: Offset of kernels.

of kernel K, relative to the top of the boundary box of the processor onto which K is mapped.

The introduction of the internal and external offset *off* 0 and *off* 1 makes the formulation much less complex. Whether or not two kernels that are mapped to the same processor conflict with each other can be determined simply by comparing the kernel's start and end times, relative to the top of the bounding box, in a similar manner to Constraint (19). The following are the constraints that show the relationship between the offset variables.

$$s_i = off_i + T * N_i, \quad \forall i \in K, \tag{20}$$

$$0 \leq off_i < T, \quad \forall i \in K, \tag{21}$$

$$off_i = off0_{proc(i)} + off1_i, \quad \forall i \in K, \tag{22}$$

$$T \geq off1_i + et(kt(i)), \quad \forall i \in K. \tag{23}$$

The constraint that guarantees that two kernels mapped to the same processor are conflict free is given as follows:

$$off1_i >= off1_j + et(kt(j)) \vee$$
$$off1_j >= off1_i + et(kt(i)), \tag{24}$$

where:

$$\forall (i,j) \in \{(k0,k1) \in (K,K) : proc(k0) = proc(k1)\}.$$

- Constraints on the number of buffers needed are derived from the inequalities presented in Section 3.2. The following constraints determine the number of producer-*DMA* buffers needed:

$$(bDMA_{i,j} - 1) * T \leq et(kt(cssrc(j))) + t_{DMA}(j), \tag{25}$$

$$et(kt(cssrc(j))) + t_{DMA}(j) < bDMA_{i,j} * T, \tag{26}$$

for $\forall i \in P, j \in S1(i)$, where $S1(i)$ is the set of source ports that originates from processor i and sends data to kernels mapped to other processors. $bDMA_{i,j}$ is 0 for $\forall i \in P$, $\forall j \notin S1(i)$.
- The following are the constraints for intraprocessor communication buffers.
  For $\forall i \in S4$:

$$(bC_i - 1) * T \leq e_{c2dst(i)} - s_{c2src(i)}, \tag{27}$$

$$e_{c2dst(i)} - s_{c2src(i)} < bC_i * T, 5, \tag{28}$$

where:

$$S4 = \{c \in C : proc(c2src(c)) = proc(c2dst(c))\}.$$

- The following are the constraints for *DMA*-consumer buffers:
  For $\forall i \in C - S4$:

$$(bC_i - 1) * T \leq e_{c2dst(i)} + t_{Ack}(i) - e_{c2src(i)}, \tag{29}$$

$$e_{c2dst(i)} + t_{Ack}(i) - e_{c2src(i)} < bC_i * T. \tag{30}$$

- The number of intraprocessor or *DMA*-consumer buffers processor i needs for source port j satisfies the following inequality:

$$bCS_{i,j} \geq bC_c, \quad \forall i \in P, \ \forall j \in CS, \ \forall c \in S2(i,j), \tag{31}$$

where:

$$S2(i,j) = \{c \in C : c2cs(c) = j \wedge proc(c2dst(c)) = i\}.$$

$S2$ is the set of channels that originates from source port j and has consumer kernels mapped to processor i.
- The minimum number of buffers processor i needs for source port j must be larger than both the number of buffers for producer-*DMA* and the number of buffers for *DMA*-consumer or intraprocessor communication. For $\forall i \in P, \forall j \in CS$:

$$\min b_{i,j} \geq bCS_{i,j}, \tag{32}$$

$$\min b_{i,j} \geq bDMA_{i,j}. \tag{33}$$

- The total amount of memory has the following constraints:

$$lm_i = \sum_{j \in CS} \min b_{i,j} * cssize(j) + lmc_i, \quad \forall i \in P, \tag{34}$$

$$\max lm \geq lm_i, \quad \forall i \in P. \tag{35}$$

- **Objective Function.** The objective function minimizes the maximum usage of local buffer.

$$\text{minimize} : \max lm. \tag{36}$$

4. Experimental results

4.1. Experimental procedure

The software pipelining approach explained in this paper is compared with the approach presented in [9]. The work in [9] presents a software pipelining algorithm which schedules streaming application onto the IBM CELL platform [10]. The overall flow of the work is described in the following:

1. Schedule kernels in the stream graph, assuming every kernel is assigned to a distinct processor. All communications between kernels are interprocessor communications. This will give the upper bound of the amount of memory used. Schedule the kernels in topological sort order, assigning the earliest stage at which the kernel can execute.
2. Estimate the buffer usage of communication between kernels, using the schedule obtained in the previous step.
3. Formulate and solve a MILP problem which partitions the kernels, so that the initiation interval is minimized, while meeting the memory usage constraint.
4. Reduce the usage of memory by pushing kernels to earlier stages. Opportunities to reduce the amount of memory used arise, since some of the communications between kernels, which were conservatively assumed to be interprocessor, are now intraprocessor communications.

#	$\|K\|$	$\|C\|$	$\|P\| = 2$				$\|P\| = 3$				$\|P\| = 4$			
			T_{min}		T_{max}		T_{min}		T_{max}		T_{min}		T_{max}	
			0.3	0.8	0.3	0.8	0.3	0.8	0.3	0.8	0.3	0.8	0.3	0.8
1	15	19	68.0	60.1	66.1	57.7	64.7	47.3	66.6	50.1	53.9	30.8	57.1	42.9
2	21	31	56.7	45.9	62.8	53.5	65.0	40.0	58.3	N/A	57.0	35.7	42.6	13.9
3	12	16	28.3	15.1	24.0	10.0	45.0	16.7	46.7	30.0	48.3	16.7	60.0	50.0
4	19	22	71.5	64.0	68.3	58.9	65.4	57.7	47.5	42.5	67.7	N/A	35.6	32.2

Figure 8: Reduction [%] in largest buffer size.

Since the approach presented in [9] and the approach of this paper have different goals (the primary goal of [9] is to maximize the throughput given memory usage constraints, whereas the approach of this paper tries to minimize the largest memory usage over all processors, given a fixed throughput), the experiments to evaluate the two approaches are conducted in the following steps to make the comparison as fair as possible.

1. In step 3 of the scheduling algorithm in [9], supply an infinite value as the memory size constraint. This will guarantee the generated schedule will have the minimum initiation interval possible. Assign the initiation interval to $T_{\min}$ and execute the remaining steps.
2. Next, go back to Step 3. This time, give the minimum amount of memory possible that will still allow the MILP solver to find a feasible solution. The problem of computing the minimum amount of memory can be solved as a bin packing problem in which items of various sizes are packed into a fixed number of bins, while minimizing the largest bin size. Assign the initiation interval obtained in Step 3 to $T_{\max}$. Execute the remaining steps.
3. Run the partitioning and scheduling algorithms explained in Section 3, using $T_{\min}$ as the initiation interval. The solver program is terminated if it does not return a feasible solution after a certain amount of time. Repeat the process using, $T_{\max}$. Compare the memory usage obtained in this step with that of Steps 1 and 2.
4. Go back to Step 1 and repeat the process until the results for all the application benchmarks are obtained.

In addition to varying the target throughput, T, the overheads of sending acknowledgment signals or data via *DMA* are varied too.

4.2. Results

Figure 8 shows the percentage of reduction in the largest buffer size, when the approach presented in this paper was taken, compared to the result obtained when the approach of the work in [9] was taken.

The first column in Figure 8 shows the benchmark application numbers. The first two were derived from the MPEG benchmark, and the third and fourth were derived from the DES and Beamformer benchmarks, respectively. All four benchmarks were taken from the StreamIt benchmark suite [4]. The execution time of each kernel was obtained by profiling.

The second and third columns, respectively, show the number of kernels and edges in the stream graphs. The number of processors in the target architecture, $|P|$, were varied between two and four. $T_{\min}$ and $T_{\max}$ are the initiation intervals explained in Section 4.1. The numbers shown below $T_{\min}$ or $T_{\max}$ (0.3 and 0.8) are the relative lengths of *DMA* transfers in proportion to the initiation interval, i.e., t_{DMA}/T. t_{Ack}/T is set to 0.1. Note that t_{DMA} and t_{Ack} can also be made functions of channels or source ports, for the algorithm to be applicable to architectures with non-uniform communication latencies.

Symphony [11] was chosen as the solver for the MILP problems. For some combinations of applications and architecture configurations, the results are not shown in the table because the MILP problems explained in Section 3.3 could not be solved in the given amount of time.

The approach of this work shows large improvements over all architecture configurations and applications. As expected, the improvement when $t_{DMA}/T = 0.8$ is not as impressive as it is when $t_{DMA}/T = 0.3$. The architecture configurations with smaller numbers of processor seem to do slightly better than those with larger numbers of processor.

The average reductions in the schedules' makespans were 56.1%, 76.8% and 63.6% when $|P|$ was 2, 3 and 4, respectively.

5. Related work and discussion

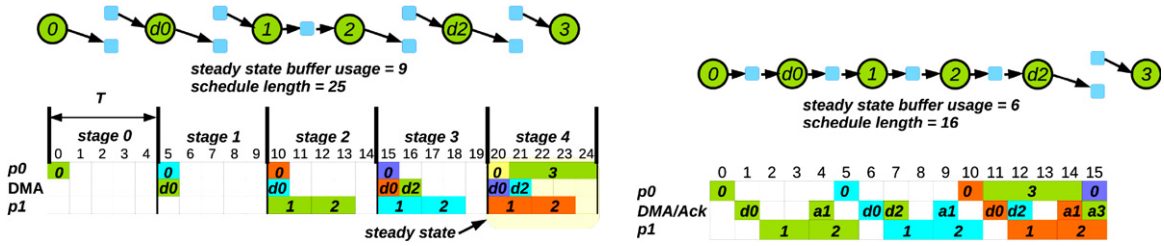
Many researchers have pursued the idea of software pipelining streaming applications onto multicore architectures. The work in [12] compiles applications written in StreamIt language onto the RAW architecture [13]. The approach exploits different types of parallelism, namely task, data and pipeline parallelism, which exist in the application in a unified manner. All communications use buffers allocated in the main memory rather than the fast local memory or cache. Also, the communication is not done concurrently with computation, but done between software pipeline stages.

Several researchers have proposed methodologies to compile streaming applications onto the CELL Broadband engine architecture [9,14,15]. Work in [14] presents a compilation technique which partitions and duplicates kernels simultaneously using integer linear programming.

The work in [15] is an extension of [9]. It presents an adaptive compilation framework that reschedules a statically scheduled streaming application based on the resources available at run time. At compile time, it formulates and solves a MILP problem, in a similar way to [9], to find a partitioning of the kernels that results in the maximum throughput, given the hardware resource constraints. The run-time system refines the partition obtained at compile time using a low-overhead variation of modulo scheduling.

The work in [16] schedules applications written in StreamIt targeting NVIDIA GPU processor. Their framework can handle streaming applications which have kernels with different rates of data consumption. The scheduling and mapping of kernels are solved as a MILP problem. The MILP problem is solved to determine whether there exists a feasible solution that meets all the dependency constraints but does not have an objective function to reduce resource usage.

The primary difference between the approach of this paper and those listed above lies in the way producer and consumer kernel pairs synchronize. The approach of previous



(a) Previous works.

(b) This work.

Figure 9: Scheduling stream graph of Figure 3.

work ensures correct synchronization between producers and consumer kernels, by avoiding scheduling kernels and *DMA* transfer processes across stage boundaries and by scheduling producer-consumer pairs in different stages. The drawback of this approach is that it tends to generate schedules that have large makespans and buffer consumptions.

Figure 9 shows the difference in how the stream graph in Figure 3 is scheduled, when the approaches of previous work and those of this paper were used. In Figure 9(a), all interprocessor communications use double buffering to prevent producers from overwriting data yet to be read by consumers, which results in a total of nine buffers being used. On the other hand, in Figure 9(b) total of six buffers are used, owing to the fact that each communication, with the exception of those between *DMA* transfer $d2$ and kernel 3, uses just one buffer.

One area of improvement of our approach will be the running time. As mentioned in Section 4, we found that our MILP-based scheduling algorithm was unable to find a solution in a reasonable amount of time when the number of PEs or kernels was high. We believe that if the inter-processor communication scheme we proposed in this paper is used, a heuristic, such as a scheduling algorithm based on list scheduling, will still be able to generate a solution that needs a small amount of buffers, although further studies will be needed to confirm our prediction.

6. Conclusion and future work

In this paper, we developed a software pipelining algorithm that schedules streaming applications onto multicore architectures. Experimental results showed a large reduction in maximum buffer size when compared with past works, exceeding 70% for some of the combinations of architecture configuration and application. The reductions in the makespans of the schedules were large as well.

In the future, we plan to develop a software pipelining algorithm that is faster than the one presented in this paper, which can handle applications with a much larger number of kernels and communication edges. We also plan to evaluate the algorithm in this paper on real hardware or simulation models.

References

[1] Kahn, G. "The semantics of a simple language for parallel programming", In *Information Processing'74: Proceedings of the IFIP Congress*, J.L. Rosenfeld, Ed., pp. 471–475, North-Holland, New York, NY (1974).

[2] Lee, E.A. and Messerschmitt, D.G. "Static scheduling of synchronous data flow programs for digital signal processing", *IEEE Trans. Comput.*, 36(1), pp. 24–35 (1987).

[3] Buck, I. and Foley, T., et al. "Brook for GPUs: stream computing on graphics hardware", *ACM Trans. Graph.*, 23(3), pp. 777–786 (2004).

[4] Thies, W. and Karczmarek, M., et al. "StreamIt: a language for streaming applications", *International Conference on Compiler Construction*, Grenoble, France (Apr 2002). [Online] Available: http://groups.csail.mit.edu/commit/papers/02/streamitcc.pdf.

[5] Kapasi, U. and Dally, W.J., et al. "The imagine stream processor", *Proceedings 2002 IEEE International Conference on Computer Design*, pp. 282–288 (Sep. 2002).

[6] Kistler, M. and Perrone, M., et al. "Cell multiprocessor communication network: built for speed", *IEEE Micro*, 26(3), pp. 10–23 (2006).

[7] Lam, M. "Software pipelining: An effective scheduling technique for VLIW machines", *Conference on Programming Language Design and Implementation*, pp. 318–328 (1988).

[8] Rau, B. "Iterative modulo scheduling: an algorithm for software pipelining loops", *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pp 63–74 (1994).

[9] Choi, Y. and Lin, Y., et al. "Stream compilation for real-time embedded multicore systems", In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pp. 210–220, IEEE Computer Society, Washington, DC, USA (2009).

[10] Kahle, J.A. and Day, M.N., et al. "Introduction to the cell multiprocessor", *IBM J. Res. Dev.*, 49(4/5), pp. 589–604 (2005).

[11] Ralphs, T. and Gzelsoy, M. "The SYMPHONY callable library for mixed integer programming", *The Next Wave in Computing, Optimization, and Decision Technologies*, 29, pp. 61–76 (2005).

[12] Gordon, M.I. and Thies, W., et al. "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs", In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 151–162, ACM, New York, NY, USA (2006).

[13] Taylor, M.B. and Kim, J., et al. "The raw microprocessor: a computational fabric for software circuits and general purpose programs", *IEEE Micro*, 22(2), pp. 25–35 (2002).

[14] Kudlur, M. and Mahlke, S. "Orchestrating the execution of stream programs on multicore platforms", *SIGPLAN Not.*, 43(6), pp. 114–124 (2008).

[15] Hormati, A.H. and Choi, Y., et al. "Flextream: adaptive compilation of streaming applications for heterogeneous architectures", In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques. Washington*, pp. 214–223, IEEE Computer Society, DC, USA (2009).

[16] Udupa, A. and Govindarajan, R., et al. "Software pipelined execution of stream programs on GPUs", In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pp. 200–209, IEEE Computer Society, Washington, DC, USA (2009).

Akira Hatanaka received his B.S. and M.S. Degrees in Electronic Science and Engineering from Kyoto University. He is currently pursuing his Ph.D. in Electrical Engineering and Computer Science at University of California, Irvine. His main research interests are parallel architectures and compilers.

N. Bagherzadeh is interested in low-power and embedded digital signal processing, computer architecture, computer graphics and VLSI design. Within the area of embedded digital signal processing, Dr. Bagherzadeh is interested in the design and VLSI development of reconfigurable processor architectures and their algorithm mapping for high-performance and low-power applications in mobile communications. This technology can be used for 3G and 4G cellular phones, as well as other telecommunications systems.

In the area of computer graphics, Dr. Bagherzadeh has been involved in the development of a new scheme for creating computer-generated three-dimensional models of a scene based on previously recorded images captured with a standard digital video camera. These models can be used for military and civilian simulator applications, as well as movie special effects.

In the area of low-power system design, Dr. Bagherzadeh has developed a software tool for scheduling and planning mission tasks to achieve power and performance objectives. This technology is targeted for planetary missions of autonomous spacecraft, as well as for unmanned military vehicles.